Run-Fail-Grow: Creating Tailored Object-Oriented Runtimes.

Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Run-Fail-Grow: Creating Tailored Object-Oriented Runtimes

G. Polito[b] L. Fabresse[c] N. Bouraqadi[c] S. Ducasse[a]

a. Univ. Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, France

b. Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, France

c. IMT Lille Douai, Univ. Lille, Informatics & Automation Dept., F-59000 Lille, France

Abstract Producing a small deployment version of an application is a challenge because static abstractions such as packages cannot anticipate the use of their parts at runtime. Thus, an application often occupies more memory than actually needed. Tailoring is one of the main solutions to this problem *i.e.,* extracting used code units such as classes and methods of an application. However, existing tailoring techniques are mostly based on static type annotations. These techniques cannot efficiently tailor applications in all their extent (*e.g.,* runtime object graphs and metadata) nor be used in the context of dynamically-typed languages.

We propose a *run-fail-grow* technique to tailor applications using their runtime execution. Run-fail-grow launches (a) a reference application containing the original application to tailor and (b) a nurtured application containing only a *seed* with a minimal set of code units the user wants to ensure in the final application. The nurtured application is executed, failing when it founds missing objects, classes or methods. On failure, the necessary elements are installed into the nurtured application from the reference one, and the execution resumes. The nurtured application is executed until it finishes, or until the developer explicitly finishes it, for example in the case of a web application. resulting in an object memory (*i.e.,* a heap) with only objects, classes and methods required to execute the application.

To validate our approach we implemented a tool based on Virtual Machine modifications, namely Tornado. Tornado succeeds to create very small memory footprint versions of applications *e.g.,* a simple object-oriented heap of 11kb. We show how tailoring works on application code, base and third-party libraries even supporting human interaction with user

G. Polito, L. Fabresse, N. Bouraqadi, S. Ducasse. *Run-Fail-Grow: Creating Tailored Object-Oriented Runtimes*. In Journal of Object Technology, vol. 16, no. 3, 2017, pages 2:1–36. doi:10.5381/jot.2017.16.3.a2

interfaces. These experiments show memory savings ranging from 95% to 99%.

Keywords tailoring; extracting; deployment; constrained devices.

1 Introduction

Deployed object-oriented applications often contain objects and code (e.g., packages, classes, methods) that the running application creates and never uses (Section 2). This problem shows itself more evident and harder to control under the usage of third-party software. Third-party libraries and frameworks are often designed in a generic fashion that allows multiple usages and functionalities, while applications use only few of them. Examples are logging libraries, web application frameworks or object-relational mappers.

Unused deployed code units have an undesired impact when targeting a constrained infrastructure. Constrained devices may have restrictive hardware such as low primary or secondary memory, or even software constraints such as Android's Dalvik VM restriction to deploy at most 65536 methods[1]. Big JavaScript mashup applications have an impact on loading time due to network speed and parsing time. These limitations may forbid the deployment of applications that contain a lot of code, or limit the amount of applications and content a user can have in his/her device.

The majority of the solutions to this problem described in the literature [RK02, TSL03, Tit06, SD10, BO14, Age96] propose to automatically detect and extract useful code, so called *tailoring*, with static call graph construction [GDDC97] (Section 6). These static approaches present several limitations:

- they are not efficient in the presence of dynamic features such as reflection, or in the absence of static type annotations;
- they do not take into account the dynamic extent of a program execution *e.g.,* they cannot reduce object graphs and metadata that appear only at runtime.

To overcome these issues we developed the *run-fail-grow* (RFG) approach (Section 3): an alternative solution to application tailoring that identifies at runtime the set of objects and code units that are actually used in an application. For such a task, in RFG we launch the application to be tailored containing all its code (*i.e.,* the *reference* application) and a second application at its side containing only a minimal set of objects and code units we want to ensure in our resulting application (*i.e.,* the *nurtured* application). RFG consists in "growing" the nurtured application into a specialized version of the reference application. For this, RFG runs normally the nurtured application, which will fail at runtime because of the absence of code and objects. RFG feeds it with the required code and objects and resumes the execution. This process repeats until the application finishes its execution or is manually stopped by a developer.

This process results in a *nurtured* application that only embeds the code and objects that were initially installed and the ones that were required at runtime. By carefully choosing initially-installed elements (so-called the *seed*), the developer can customize the scope of the tailoring process making possible different levels of tailoring.

[1] According to Dalvik's bytecode documentation (`http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`.), the source register accepts values between 0 and 65535.

The dynamic nature of our solution allows its usage in dynamically-typed languages, and applications using reflection. It also guarantees that unused runtime metadata and objects are tailored if not used by the application. Our solution does not require any modifications to the original application, and it is therefore applicable to legacy code. This paper makes the following contributions:

- We present a run-fail-grow technique which is applicable to dynamic languages with no type annotations and has natural support for reflection.
- We introduce the customization of such an approach by specifying a *seed*.
- We present a prototypical implementation called Tornado based on Virtual Machine modifications (Section 4). We use Tornado to validate our approach with several experiments. Our experiments show promising results (Section 5) *e.g.,* memory savings ranging from 95% to 99%.
- We provide a detailed classification and comparison of existing related work on the area of tailoring.

2 Motivation: Software Bloat

Software applications are deployed as a set of libraries and executable programs containing code. Such code is internally organized in different *code units*.

Definition (Code Unit)**.** *A code unit is a software element that represents an essential concept of a programming language. For example, in an object-oriented programming language, code units are classes and methods. A language adding functional extensions has also functions as code units.*

When a program is executed, we can identify among these code units *dead code units*.

Definition (Dead Code Unit)**.** *A dead code unit is a code unit that is never used during a given set of executions of a program. For example, a class that is never instantiated is a dead class; a method that is never called is a dead method.*

This problem appears as well with objects, although objects are created dynamically during the program execution. We call these *dead objects*.

Definition (Dead Object)**.** *A dead object is an object that in a given scenario is not used but still not garbage collected because it is referenced [MPBD+10, MPBD+11b]. We consider as unused those objects that are never sent messages or whose state is never accessed.*

To establish a common vocabulary on the rest of this paper, we define also *program unit* and *dead program unit* as follows:

Definition (Program Unit)**.** *A program unit is either an object or a code unit.*

Definition (Dead Program Unit)**.** *A dead program unit is a program unit that is not used during a given execution of a program.*

These dead program units cause applications to occupy more memory (primary and secondary) than the application really requires to be executed. This problem is already defined in the literature as *Software Bloat* or more specifically *Memory Bloat* [XMA+10, BNGG11].

Software bloat represents a serious drawback in constrained devices. First, unused program units may forbid the deployment into a constrained resource device by requiring more resources than available. It may interfere with the deployment and usage of other applications, because of large memory footprints in both secondary (disk storage) and primary (RAM) memory [MP12] or the presence of slow networks in the case of rich web applications. Second, some deployment targets may have an infrastructure designed in such a manner that it forbids the deployment of large applications. For example, the former Android's VM, Dalvik, restricted applications to deploy at most 65536 methods.

2.1 A Motivating Example

To clearly show the problem, consider the application illustrated by the UML diagram in Figure 1 and the source code in Figure 2, written in the Pharo Smalltalk language[2]. This application contains a MainApp class with a start method, which is the entry point of our application. The start method creates an instance of StdoutLogger and logs the application start and end. In turn, the StdoutLogger uses the stdout global instance to log in the standard output the current time and the message. To print the time, the StdoutLogger makes use of the Time class from the base libraries of the language. Note that for the sake of clarity, we didn't include in the example all base libraries, although in modern programming languages they represent a large codebase with several features going from networking to multithreading. For example, Java 8 SE contains 4240 classes[3], and the development edition of Pharo 3.0 [BDN+09] contains 4115 classes and traits.

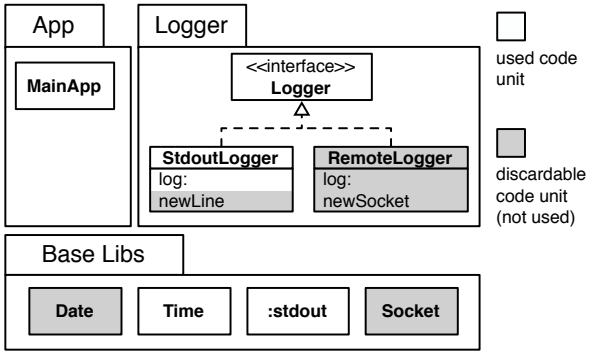

Figure 1 – **Example of unused code units.** In gray, the unused code units that can safely be removed.

This application, written in a reflective language such as Pharo, suffers from both kind of software bloat: dead code units and dead unused objects.

Unused Code Units. The avid reader can for sure identify the unused code, shown in grey in Figures 1 and 2. Unused classes such as RemoteLogger or Date are not used in the application regardless they are part of our package or a third-party package. Some methods such as

[2]To those not versed in Smalltalk-like syntax, these are the equivalents to Java that are required for this example:

```
SomeClass new.              -> new SomeClass();
self simpleMethod.          -> this.simpleMethod();
other methodWithArg: arg.   -> other.methodWithArg(arg);
"a comment"                 -> /*a comment*/
'a string'                  -> "a string"
```

[3]according to the javadoc API.

```
1  MainApp»start (
2      logger := StdoutLogger new.
3      logger log: 'Application has started'.
4      "do something"
5      logger log: 'Application has finished'. )
6
7  StdoutLogger»newLine (
8      stdout newLine. )
9
10 StdoutLogger»log: aMessage (
11     stdout nextPutAll: Time now printString.
12     stdout nextPutAll: aMessage.
13     stdout newLine. )
14
15 RemoteLogger»log: aMessage (
16     | socket |
17     socket := self newSocket.
18     socket nextPutAll: Time now printString.
19     socket nextPutAll: aMessage.
20     socket newLine. )
21
22 RemoteLogger»newSocket (
23     "...."
24     "creates an instance of socket given some configuration" )
```

Figure 2 – **Code of the example logging application.** In gray, methods not used by the application.

newLine (lines 7-8 of Figure 2)

of the StdoutLogger class are never called.

Unused Objects and Runtime Metadata. Object references stored in classes prevent objects from being garbage collected. For example, a singleton object may have been created as the result of a singleton pattern but never really used in the application. Moreover, reflective languages such as Pharo store metadata at runtime in dedicated data structures, usually as class state or global state. This runtime information is important for reflection and other meta-programming facilities. For example, Pharo allows iterating the methods of a class or even a class hierarchy as follows:

```
1 Collection methods.    => { #do: . #select: . #collect: . #size . ... }.
2 Object allSubclasses. => { Collection . Integer . Compiler . ... }.
```

In this example, most of the reflective metadata such as class names, superclass-subclass relations, and others that we can found in specific programming languages such as Pharo (*e.g.,* Traits [SDNB03] and Slots [VBLN11]) are not needed at runtime by this program. We can think about stripping and specializing this runtime metadata as well as the program code units.

2.2 Challenges of Application Tailoring

We would like to generate a new version of this application not containing these dead program units while keeping the application's behavior. We call this technique *application tailoring*. A lot of work exists on the tailoring of statically-typed applications [CGV10, RK02, TSL03, PRT+04, TP01], where type annotations aid in the resolution of which piece of code will be used at runtime. However, static analysis is not an efficient option in the context of dynamically-typed languages, nor in the presence of meta-programming and reflection [LWL05]. In this context of dynamically-typed and object-oriented programs that may use reflection, we identify the following main challenges for detecting dead program units:

Dynamic typing. Analyzing dynamically-typed languages is challenging because of the absence of static type annotations. This makes dynamically-typed languages difficult to analyze, impacting on the effectiveness of the analyses and increasing false positives. For example, given the expression "anObject foo", an analyzer cannot easily determine the type of anObject, since it may reference objects of different types at runtime. Also, it may confuse many different implementations of foo, not necessarily related to each other. Moreover, techniques like doesNotUnderstand: allow developers to respond at runtime to unanticipated messages, which were not statically defined.

Polymorphism and inheritance. Polymorphism in object-oriented languages allows an object to treat objects of different concrete types in the same way as soon as they share a common interface. Inheritance plays a similar role: any class can extend another class and provide different behavior while sharing a common API. As a consequence, both polymorphism and inheritance make the behavior of a program more difficult to predict by just statically analyzing its code units. Usually their resolution is delayed until execution time, when dynamic type information is available [TGP89, DDG+96].

Application runtime configurations. Modern applications often contain libraries and frameworks besides their proper code. To make these different code units fit together, applications rely on heavy configurations. These configurations are usually present in configuration files looked up dynamically by the application. Based on these configurations, the dependency injection pattern is usually used to dynamically set up the application. This recurrent and standard process for configuring applications implies that static analysis will be inefficient to detect used program units without library-specific knowledge.

Reflection. Reflection makes static analysis inoperative by allowing an application to execute unanticipated pieces of code [LWL05]. Any String resulting from a program execution or program configuration can denote a message send[4], the name of a class to be instantiated, or even a script to be executed. Reflection is indeed important to cover, since it is a broadly used tool in industrial applications with object relational mappers such as Hibernate or Glorp and web frameworks such as Ruby On Rails, Struts or Seaside. Existing research tries to overcome this limitation by detecting usages of reflection and help developers to transform the code and make it statically analyzable [BSS+11]. This remains however an open research topic.

[4]We refer method invocations as message sends to represent the dynamic property of the invocation.

2.3 Evaluation Criteria for Application Tailoring

To understand the design space of application tailoring, we studied existing solutions in the literature (cf. Section 6) and we defined a criteria to evaluate such solutions and objectively compare to our solution. This criteria focuses on the applicability of such approaches, answering the following questions: What parts of an application does it tailor? What changes should I do to the tailored application? Based on these questions, we arrived to the following criteria.

Reflection Awareness. An ideal tailoring solution should handle correctly reflective code and resolve the unanticipated code executions in the same way as the application would do during runtime.

Base-library Specialization. A programming language provides base libraries covering common and generic tasks. Not every program unit in these libraries is used in an application. An ideal tailoring solution should tailor base libraries to reduce an application's deployment memory footprint.

Third-party Library Specialization. Applications use third-party libraries and frameworks covering several aspects of application development such as user interfaces, persistence or publication of services. Third-party libraries contain large code bases and many dependencies. Thus, an ideal tailoring solution should consider the existence of third-party software.

Applicability in Legacy Code. An ideal tailoring solution should be applicable on already existing applications and not require modifications to them.

Standard Deployment Infrastructure. An ideal tailoring solution should produce a version of the application that is able to run on the official production infrastructure (such as the VM) without overhead.

Configurability. An ideal solution for tailoring an application should support many different levels of application. Some applications may not need to tailor base libraries because they are shared with other applications. However, tailoring base libraries may be useful on applications deployed alone in constrained devices.

Applicability without Type Annotations. An ideal tailoring solution should be applicable to dynamic languages with no type annotations.

Completeness. An ideal tailoring solution should guarantee that all code units selected as part of the deployable application are those needed. That is, it does not contain extra program units, nor it misses program units.

3 Our Approach: Run-Fail-Grow

We present the approach, then illustrate it and discuss the points mandatory to detect missing program units as well as the notion of seed.

3.1 Run-Fail-Grow in a Nutshell

We propose a run-fail-grow (RFG) approach to tailoring. Briefly, RFG works by launching a *reference* application encompassing the full application with all its program units (base libraries, third-party libraries and application code) and a *nurtured*

application that contains initially a seed *i.e.,* a potentially empty set of program units. We execute the nurtured application *entry point* as an application thread. The execution of such thread will proceed to install program units from the reference application on demand, when *missing* program units are detected. During this process, the reference application (and all its threads) remain suspended to avoid side-effects that would alter the nurtured application. The process finishes when all nurtured application threads finish or when the developer stops them explicitly. Figure 3 depicts the basics of our run-fail-grow approach.

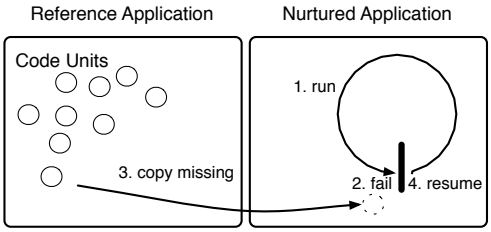

Figure 3 – **Application tailoring with a run-fail-grow approach.** We (1) run the nurtured application and (2) detect the missing units on failure. For each failure, (3) we copy missing program units from the reference application and then (4) the execution is resumed (just before the failure point) until the process finishes.

In RFG, we can beforehand initialize the nurtured application with a *seed*. Seeds are useful to ensure the availability of specific program units in the final application. Also, seeds serve the purpose of managing the scope of tailoring. Since program units are installed on demand, already available program units will not be subject to tailoring. For example, a seed containing all base libraries will affect only those program units that belong to the application code and third-party libraries. By using an empty seed, tailoring will also affect base libraries (cf. Section 3.5).

The key point of RFG is to detect used (and missing) program units at runtime, when all the required information for tailoring can be extracted from runtime information *e.g.,* the exact type of receiver and arguments objects, the exact class hierarchies. The usage of runtime information makes RFG usable in the absence of *static type declarations* and makes it easy to handle *polymorphism, inheritance and reflective operations. Application configurations are honored* since the code that reads and interprets them is actually executed, without the need of custom code for them. *Reflection is supported for free* since reflection invocations are treated as simple message sends and executed as any other code, and strings composed dynamically by the application are available at runtime.

The main drawback of this approach is that it requires that application entry points exercise all important paths of the application execution to ensure application completeness, otherwise the tailored application may not contain all program units required in future executions. In our comparison with related work in Section 6 we show that all existing approaches also suffer from similar problems due to reflection and other dynamic properties of applications. In Section 7.1 we discuss about possible alternatives to overcome this issue.

3.2 Run-Fail-Grow by Example

We illustrate in this section how RFG works with the example application introduced in Section 2.1. For the sake of clarity, we show how RFG tailors the application

program units and we avoid the tailoring of base-libraries. In RFG terminology, the base libraries are included as part of the seed.

Setup of the Environment. First, we launch the reference application (cf. Figure 4) and the nurtured application (cf. Figure 5 Step 0). We fill the nurtured application with a seed containing the language base libraries. Thus, each application has its own copy of the base libraries, as shown in this case with the Date and Time classes and the stdout object.

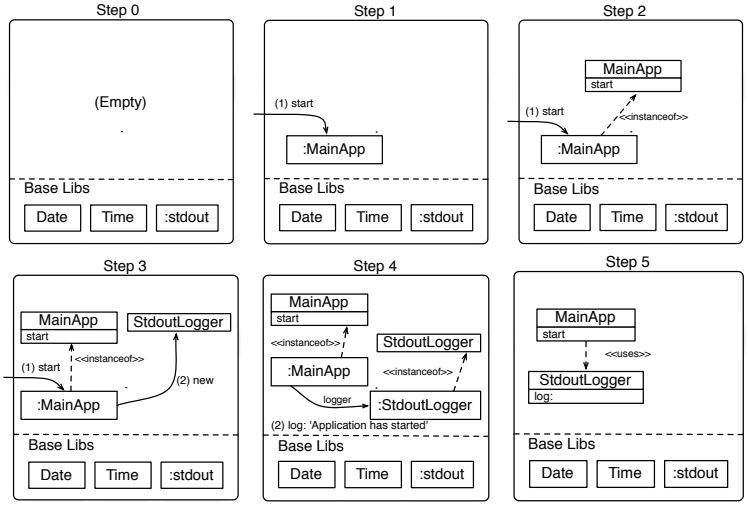

Figure 5 – **The nurtured application at different steps of tailoring.**

Install the application entry point. In the nurtured application, we install our application entry point *i.e.,* a MainApp instance (aMainApp) and a process that will execute the statement "aMainApp start" (cf. Figure 5 Step 1). Note that although we are referencing an instance of the class MainApp, the MainApp class is not installed yet.

When the execution starts, the mainApp instance receives the start message, and we detect the MainApp class and its start method as a missing program unit failure. We install these two missing program units (cf. Figure 5 Step 2) and finally the MainApp»start method is activated and starts running.

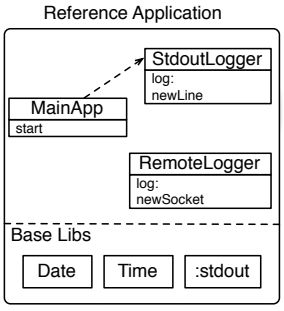

Figure 4 – **Reference application encompassing all program units.**

Activating the start method. The method start defined in Figure 2 is executed, as we can see in Figure 5 Step 2. During the execution of its first statement (line 2 Figure 2) we detect a missing program unit failure: the StdoutLogger class does not exist. Thus, before continuing, we install a StdoutLogger class with the same number of instance variable declarations as its reference version (cf. Figure 5 Step 3). This class does not contain, however, any methods nor metadata (*e.g.,* superclass, package,

subclasses) from its reference counterpart since they are not necessary. Once we have installed the StdoutLogger class, we resume the execution. The first statement results into a new StdoutLogger instance.

During the second statement execution (line 3 Figure 2), we detect a missing program unit failure on the log: message (cf. Figure 5 Step 4): the corresponding method is not installed in the StdoutLogger class. We install the method inside the corresponding class and resume the execution. This time the method is found, and the log: method is executed.

Once the log: method finishes, the execution returns to the start method. There, the third statement (line 5 Figure 2) is executed with no intervention of our technique, since the log: method is already available. Figure 5 Step 5 shows the final state of the nurtured application: it contains only the methods and classes that are actually used by the application. Leaf objects used during the process have been garbage collected.

3.3 Detecting Missing Program Units

RFG needs to notice at runtime missing program unit failures. RFG's algorithm is based on traps to achieve this task, as shown in Algorithm 1.

```
Initialize reference application;
Initialize nurtured application with its seed;
Install entry point(s) and their traps;
while not finished do
    run the nurtured application;
    if trap was activated then
        stop execution;
        install missing program units and their traps;
        resume execution;
    end
end
```

Algorithm 1: An abstract view of the run-fail-grow process.

Traps are placeholders that are installed in the nurtured application in the place of real program units. They are triggered whenever the application accesses them. In case a trap is triggered, we stop the nurtured application execution, we install the missing program units replacing their corresponding traps (and making sure that new traps are installed), and finally resume the execution from the moment immediately before the trap was triggered. Traps are installed dynamically in the nurtured application following the information flow of the application *e.g.,* when a method A is installed some traps are installed on it to capture possible missing program unit failures it may cause. We identified the following as the basic traps that are necessary to tailor an application:

Missing class trap. A *missing class* trap captures messages sent to objects whose class and/or superclass does not yet exist inside the nurtured application. This situation can happen when the reference application contains already created objects through global well-known locations (*i.e.,* static and class variables, global variables) and the execution of the entry point execution does not lead to re-instantiate such object but rather accessing it. When RFG finds one of these traps, it installs the corresponding class. We refer to this class as a *partial* clone (or empty shell) of the missing class *i.e.,* all its internal state (class metadata and user-defined *class/static* variables) is initialized with placeholder traps. These

traps capture further accesses to the class state.

Missing method trap. A *missing method* trap captures method invocations whose methods are not defined in the nurtured application yet. When the application execution triggers one of these traps, RFG installs the corresponding method in the class hierarchy of the object. In case of missing classes, for example if the invoked method belongs to a (non-installed) class found up in the hierarchy, RFG installs them too. Missing method traps capture also overridden methods, Section 3.4 discusses the importance and subtleties of this point.

Missing object trap. A *missing object* trap captures messages sent to objects that do not yet exist inside the nurtured application. These traps are found, for example, in global well-known locations (*i.e.,* static and class variables, global variables). When RFG finds one of these traps, it installs the corresponding object. The object installed is a *partial* clone (or empty shell) of the original object *i.e.,* its instance variables contains placeholder traps to capture the access to its class and fields. These traps capture further accesses to the object state.

3.4 Correctly Managing Method Overrides

Method overrides require careful consideration. Let us take as an example the case in Figure 6. In the reference application, a class hierarchy includes classes A and B, B is a subclass of A. Both classes contain a method doSomething, thus the method in class B overrides the one in class A. Now, let us consider that class A and its method doSomething are installed in the nurtured application by RFG. RFG must place method traps in override sites to avoid changing the semantics of our application, othewise upon trying to invoke method doSomething on an instance of B, the method lookup will find and execute A's doSomething instead of B's.

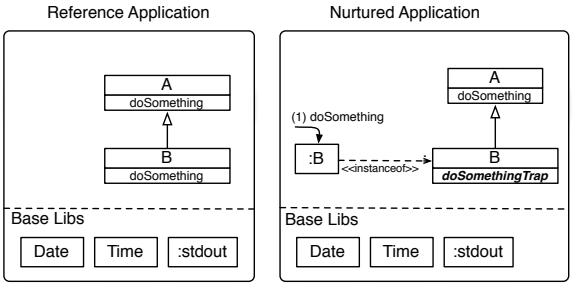

Figure 6 – **The need of overriding traps.** Method traps should capture the overridden doSomething message send to avoid the superclass method to be executed wrongly instead of the subclass method execution.

3.5 Customizing Dead Code Elimination with Seeds

The level of tailoring of RFG can be specified using a seed:

Definition (Seed)**.** *A seed is a set of non-conflicting program units that are installed into the nurtured application before its execution.*

Program units present in the seed are available for the nurtured application during its execution. Therefore, their usage does not trigger missing program unit failures during the application execution. This makes seeds useful to cover different tailoring scenarios. Let us take as a first example a smartphone where the base libraries of the language are already available, so they are shared amongst the many applications

installed in it. When targeting such a smartphone, base libraries are already present and we do not need to produce a specialized version of them, but specialize only third-party libraries and application code. In this case, we use a seed providing the language base libraries. Let us take as a second example a constrained robot-like device which will contain only our application. When targeting this robot as deployment scenario, we want to specialize all the code to deploy including base libraries. In such a case, the seed is empty to allow the RFG algorithm to work on every program unit.

Figure 7 illustrates how the scope of RFG is limited by the usage of seeds in two different scenarios. We use in both scenarios the same application containing program units corresponding to the base libraries, third-party libraries and application program units. In the first scenario (Application I) the seed covers base and third-party libraries, thus RFG applies and selects a subset of the application program units only. In the second scenario (Application II), the seed covers only base libraries, thus RFG applies and selects a subset of the program units from the third-party libraries and application code. We call this picture a *tailoring map*. Tailoring maps show in gray program units provided by a seed. Program units that are subject to RFG are shown in white. The thick black stroke illustrates the program units selected and installed by RFG in the nurtured application.

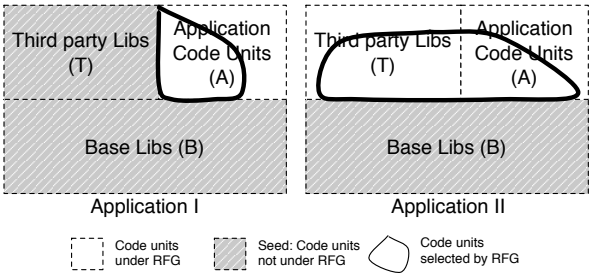

Figure 7 – **Tailoring Map.** A tailoring map describes which program units of an application are included in the seed (in gray), which ones are subject to the RFG technique (in white) and which ones are finally selected (within the thick black stroke).

4 Tornado: A RFG Implementation

We now describe the general architecture and building blocks of Tornado, our implementation of the run-fail-grow approach. We start by analyzing the requirements for this implementation, and the strategies for missing program unit detection mentioned in the previous section. Our implementation can be found in `http://smalltalkhub.com/#!/~Guille/ObjectSpace/`.

4.1 RFG's Implementation Requirements

We identify the following requirements for a development platform to implement RFG. Some platforms present all of these elements while some others present only part of them. In the latter case, the missing elements should be developed as part of the solution. In this and the following sections we explain how we fulfilled each of these requirements and how we put them together to implement our solution on the Pharo programming language. Note that these features are only needed to implement RFG and prepare an application for deployment. Once RFG is applied, we must be able to deploy our application on the standard platform infrastructure (virtual machine, operating system), without special support.

Isolated application environments. RFG requires executing both the reference and the nurtured applications in separate isolated environments. In general

terms, these isolated environments should prevent name clashes and avoid each application to interfere with the execution of the other one.

Control application execution. RFG requires full control on an application execution. It needs to be able to suspend all threads of an application when a trap is triggered, and to resume them once the trap is handled.

Capture message sends. RFG requires to intercept an application execution at runtime to detect missing program unit failures, and thus, to implement traps. In particular, it needs the ability to intercept all message sends of the application, and in particular method invocations.

Install and Query Code at Runtime. RFG requires a platform where it is possible to install code and query the installed code at runtime. Classes, methods and objects have to be installed at any moment of the execution, including the modification of classes that already contain instances, or objects that are already cloned. Also, we need to fetch the program units installed in the reference application.

4.2 Tornado's Overview

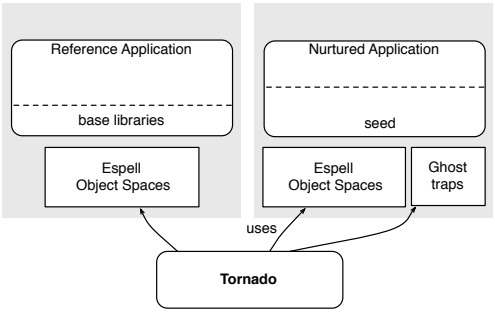

Figure 8 – **Tornado's architecture overview.** Each application (reference and nurtured) are located inside an Espell object space (in gray). Tornado controls both applications through Espell runtime manipulation interface. Traps are installed into the nurtured application with the Ghost library.

We implemented our RFG technique as a tool called *Tornado*. Tornado is implemented using the Pharo programming language, to tailor applications written in this same language. Pharo is a reflective and dynamic programming language inspired by Smalltalk [BDN+09]. Tornado's architecture is based on two main components: Espell object spaces (cf. Section 4.3) and Ghost proxies (cf. Section 4.4). A Tornado environment is illustrated in Figure 8. Tornado initiates and pauses the reference and nurtured applications and controls them *remotely*. It installs traps on the nurtured application using Ghost proxies, and uses Espell to control execution and query and install program units into them. In the following, we detail how we fulfilled each of RFG's requirements:

Process reification provides execution control. Pharo provides support for execution control in its runtime. It reifies processes and method activations (*i.e.,* instances of Process and MethodContext respectively) allowing one to manipulate them as simple objects. However, Pharo does not provide the ability to isolate two application executions. Espell object spaces cover this missing feature.

Advanced proxies. Pharo's libraries include Ghost [MPBD+11a, PBF+15], an advanced proxy implementation. Ghost allows one to capture any kind of message sends, intercept particular method executions, and even to capture usages of classes and special objects of the runtime. We use Ghost to implement all our execution traps.

Limited scope of reflection. Pharo is a dynamic language inspired by Smalltalk, and as such, it allows one to introspect and modify the runtime entities through reflection. However, reflection in Pharo is limited only to the running environment. RFG requires to reflect on two different environments: the reference and the nurtured applications. On the one hand, it needs to introspect the reference application to obtain the code units to install in the nurtured application. On the other hand, it needs to introspect and modify the nurtured application to install program units. Espell object spaces extend the reflective capabilities of Pharo and introduce the ability to introspect and modify two different environments.

4.3 Object spaces: An object runtime manipulation interface

Tornado controls the execution of the nurtured application and manipulates it at runtime using Espell object spaces [Pol15]. Espell offers an *object runtime manipulation interface* that we developed for the Pharo programming language. Espell comes with a library and an extended virtual machine that allows the manipulation of the runtime system of a Pharo application: control its execution, install code on demand and query it. Using Espell, a Pharo application is confined within a *protection domain* that we call an *object space*. The Espell library presents a first class representation of an object space which serves as a high level API to manipulate those protection domains.

In our Tornado implementation, the reference and nurtured applications are contained each in a different object space. Tornado places traps inside the nurtured object space and starts its execution. This execution is performed directly on a Pharo Virtual Machine, and thus, there is no speed overhead as long as traps are not involved. Whenever the nurtured application execution finds a trap, it pauses and returns the control to Tornado. Tornado inspects the classes and methods in the reference object space through mirrors [BU04] and installs the needed program units from the reference object space on demand, either by creating new objects or compiling new methods. Then, it resumes the nurtured application execution from the message send that activated the trap.

4.4 Execution Traps with Ghost Proxies

Implementing execution traps such as the ones described in Section 3.3 requires a powerful intercession library. Traps must capture *all* message sends to objects provided by the language runtime as well as the application objects, including classes (for example for the case of class messages or static methods). They must capture *this/self* and *super* message sends as well as method overrides.

We implement traps as proxies, using the Ghost proxy library [MPBD$^+$11a]. Ghost proposes a low-memory footprint, general-purpose proxy implementation for the Smalltalk language supporting the creation of proxies for normal objects as well as classes and methods. In this paper we use the term proxy in two different senses. First, we use Ghost proxies in the sense of the original GoF proxies [GHVJ93], where a proxy is a placeholder representing an external resource. In this sense, a proxy does not co-exist with the object it represents. The object the proxy represents exists in a different address space, inside the reference application. Second, we also use Ghost proxies in the modern sense of meta-objects to control message sends, such as JavaScript proxies [VCM10]. Figure 9 illustrates how our running example would look like at runtime with the existence of ghost proxies.

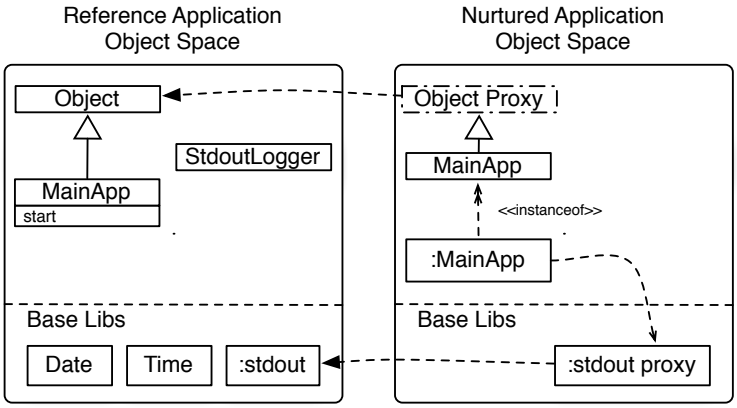

Figure 9 – **Tornado running example.** Tornado represents traps as placeholder proxies representing their corresponding reference objects. Proxies are replaced by copies of the reference objects as soon as they are used in the application execution.

Ghost proxies allow the detection of all situations corresponding to our traps. Tornado handles a table relating each proxy to the program unit it represents in the reference application. Additionally, each proxy is attached to a *handler* that may perform some action when the proxy receives a message. We rely on this concept to perform the right action for each trap. We discuss below the different kinds of proxies and handlers we use and how they support RFG.

Missing object/class trap. In the Pharo language, every class is an object, therefore all mechanisms that apply to objects apply to classes too. Consequently, our implementation of missing object traps also fulfills the role of missing class traps. We implemented this trap as an object proxy triggered when the proxy receives a message. Its handler replaces the proxy by a copy of the original object from the reference application. The copy is created, and all references to the proxy are replaced by references to this new object, which is achieved through the become: facility of the Pharo language, which dynamically swaps object references. Each field and the class of this new installed object are installed as new missing object traps.

Missing method trap. We implemented the missing method trap in Tornado as a class proxy located at the top of the class hierarchy. Whenever a message is sent to an object, the VM looks up the method in the class hierarchy of the object. This trap is triggered when a message arrives to the top of the hierarchy, meaning that there was no method for it in the hierarchy. When triggered, the handler first installs all classes of the hierarchy up to the class defining the method, and then it installs the method in the corresponding class. If no method was found to install, Tornado sends the doesNotUnderstand: message (an equivalent to *e.g.,* Ruby's method_missing and Python's __getattr__) to honor the dynamic semantics of Pharo.

Missing override trap. We implemented missing override traps in Tornado using method proxies. Method proxies are placed in the method dictionaries of classes containing overridden methods, taking the place of the original method. When

Tornado installs into the nurtured application a class defining one or more overrides in the reference application, it installs into this class a method proxy for each of these overrides. When the method lookup finds a method proxy in the method dictionary, it triggers the execution of the trap. The handler of this trap compiles a new method with the same source as its reference method and then installs it inside the nurtured application.

Primitive method trap. Primitive method traps are specific to the implementation of Pharo. Pharo's primitive operations such as number arithmetic are implemented through primitive methods. Primitive methods are implemented in the Virtual Machine and may directly access the fields of the primitive arguments by forging references and directly manipulating memory. By doing this primitives bypass our traps: when a *missing object trap* proxy is the argument of such a method the VM can silently modify this proxy, without activating the trap. To solve this, we introduce special primitive method traps, method proxies that decorate Pharo's primitive methods. When a primitive method is executed, the trap is triggered and its handler triggers each of the missing object traps received as arguments, if any. This is how Tornado forces the installation of the arguments and the primitive is executed with actual objects instead of proxies, as expected.

4.5 Object Installation and Propagation Rules

As we explained before, Tornado installs all objects inside the nurtured application on demand, as *partial copies*, *i.e.,* the objects referenced by the original object will not be copied along with it by default, but traps replace them. When Tornado installs an object inside the nurtured application, this new object has the same format and size as its original counterpart. To better control this behavior, a trap has attached a *Propagation rule*. This propagation rule determines how the object fields are propagated on installation. Tornado provides the following propagation rules to customize installation:

Missing object trap. This is the *default propagation rule* and end-user applications can usually be tailored with just them. This propagation rule installs a missing object trap in each field of the object that is being installed.

Materialization. This propagation rule forces the installation of the object referenced by the field. This is used for those cases where we need to ensure that some structure is present for the Virtual Machine *e.g.,* the first three fields of class objects (superclass, format and method dictionary) cannot be proxies because they are used by the VM for method lookup. The same happens with other objects reifying low-level concepts such as activation records or semaphores.

Substitution. This propagation rule forces the reference of the object installed to be replaced by another object reference. The usual use case of this rule is replacing some object reference by nil, the undefined object in Pharo, and so force lazy initializations.

4.6 Object Identity and Proxies

Tornado takes care of the identify of the program units using an identity table. Such an identity table is important because Tornado needs to avoid installing twice the

same program unit. Otherwise, duplicated program units could cause problems in applications relying on object identity.

Identity is also important to preserve in the presence of proxies. Tornado guarantees that identity checks always preserve object identity by enforcing the following invariant: *An object and its proxy do not exist concurrently in the nurtured application.* That is, the nurtured application contains either the object or its proxy, but not both at the same point in time. When the proxy is replaced by the copy of the reference object, all references to the proxy are replaced by references to the new object. The proxy is no longer referenced and thus, garbage collected. This invariant guarantees that identity checks that should be true will indeed be true because either the compared references point both to the same proxy, or both to the same copy.

4.7 Implementing Seeds in Pharo

A seed is in charge of initializing the nurtured application object space with the elements we want to ensure in it. Our current implementation supports two ways of describing and building seeds:

Loading an already existing memory snapshot. The nurtured application object space is initialized by loading an already existing snapshot containing classes, methods and objects. This technique consists in using a memory dump from an object heap containing all the classes and objects desired in the seed. This memory snapshot should follow Pharo's object format.

Creating all seed program units from scratch. The nurtured application object space is initialized with objects built from scratch. This technique uses a bootstrapping process [PDF+14]. With bootstrapping, we describe declaratively the contents we want in the seed and we build it automatically.

4.8 Preparing the Application for Deployment

Once the different entry points finish their execution or the developer manually stops Tornado's process, we proceed to prepare the application for deployment. Ideally at this point, the nurtured application contains all the program units needed to run. Tornado removes all leftover traps and extracts the nurtured application. Tornado identifies the traps by the presence of proxies and replaces the references to those proxies by references to another object, defaulting to the nil object. Proxy objects do not then represent a drawback in space consumption because they are garbage collected. Once the traps are removed, the nurtured application keeps no dependencies to Tornado. Thus, the application can run outside the Espell infrastructure with no performance penalties.

Finally, Tornado extracts the application program units using one of two different techniques: (a) the creation of a snapshot file containing all program units and already initialized objects; or (b) build a static description of the application containing the code for all classes and methods that should be part of it.

5 Experiments and Results

We know present the methodology and the experiments we conducted to assess our approach and its implementation.

5.1 Experiment Methodology

We evaluated RFG with Tornado by conducting five experiments that tailor different Pharo applications, with increasing requirements. We chose our experiments with the objectives of (a) understanding how minimal are the applications we can tailor, (b) exploring how successfully we address the challenges we stated in Section 2.2 and (c) exercising those cases that push to the limit the interaction between the language and the VM. Each of our experiments is detailed in the following sections.

Our experiment methodology consisted of the following steps:

1. **Set up the seed.** Most of our experiments use what we have already called an *empty seed*. This seed is, however, not completely empty but contains some minimal infrastructural objects that are needed for language-VM interaction, and is therefore 10KB large. Our last experiment, the largest one, evaluates first the usage of an empty seed and second a seed containing the language base libraries.

2. **Execute the application.** This step consists in installing and executing the entry point processes of our application. In particular in our last experiment (an interactive web application), we interact with our application through a web browser. We let the application run until all its entry points are finished. The last experiment, a web application, is the sole exception to this: we stop Tornado manually once we have interacted with all the application features.

3. **Extract the application.** We extract with Tornado the resulting application by making a snapshot of it in a Pharo binary image file. We test the generated snapshots to verify that they work properly, either by using the application or debugging them when they involve no I/O. We evaluate the behavior of the tailored application under the assumption that only the features used during tailoring should work.

4. **Perform measurements.** We measure the size of the generated snapshots files and compare them against two different Pharo distributions prepared for production. First we compare the obtained measurements (if possible) with an experimental shrunk version of the Pharo distribution called *PharoKernel*. PharoKernel was developed independently from us by Pavel Krivanek. We make another comparison with the size of the official Pharo distribution prepared for production. Pharo allows one to prepare a snapshot for production, cleaning up some caches and removing some well known objects and classes from the system, thus, freeing space.

5.2 Experiment I: Adding Two Numbers

The smallest (in terms of size) interesting program to tailor is adding two numbers, without the involvement of any I/O *i.e.,* an application just executing the `"2 + 3"` statement as entry point. Tailoring this program is challenging because it stresses the infrastructure by installing only the minimal elements an application needs to run. It makes evident how small a tailored application can be. Additionally, it is interesting since it makes use of the following features of the Pharo language and infrastructure:

Immediate objects. Immediate objects are objects encoded in the object reference instead of being allocated in the heap. Immediate objects do not contain a

reference to their class in the object header, as there is no object header. Instead, the object reference where the object is encoded contains a bit tag that the VM uses to identify the immediate object. This means that the Pharo VM must be configured with the immediate object classes (or their proxies) to send messages to these immediate objects. In this experiment we use immediate small integers, instances of SmallInteger.

Special selectors. The method selector + is a special selector for the Pharo VM. Special selectors are optimized as they are broadly used messages, for example for arithmetics. First, they are implemented as special bytecodes to avoid method lookup. If the special bytecode cannot be executed because some VM assertions are not valid (*e.g.,* class and object format assumptions), the VM performs the default method lookup. In this experiment the VM should take care of small integer arithmetic *i.e.,* it should fulfill all VM assumptions and not perform a method lookup; Tornado should install no extra methods nor classes.

5.3 Experiment II: Factorial of a small number

The following experiment in incremental complexity is the factorial of a small number, again without the involvement of any I/O *i.e.,* an application executing the "10 factorial" statement as its entry point. Factorial uses arithmetic as the previous experiment (sums and multiplications), while it also adds the following interesting cases:

Method lookup. The factorial message is sent to a small integer but not optimized as it is not a special selector. Thus, the VM looks up the corresponding method up in its class hierarchy. The method factorial is defined in its superclass (Integer).

Recursion. The factorial implementation in Pharo base libraries is recursive. Additionally, this recursion activates the factorial method many times, creating many activation records in the VM. When there is a stack overflow in the VM's stack, the VM does not stop the execution: it instead reifies activation records as objects in the heap and frees the stack to continue the execution. To do this, the VM has to correctly be configured with the class used to instantiate activation records.

5.4 Experiment III: Factorial of a large number

We experimented with an application whose entry point was the "100 factorial" statement. This application does not make use of any I/O either. The factorial of a large integer eventually creates integers that exceed 32 bits, and thus, do not fit as immediate small integers. This experiment adds the following interesting cases:

Large integers. Large integers in Pharo are represented, in contrast to immediate small integers, as objects allocated in the heap with their own object header and arbitrary length. Large integers are created automatically by the VM when the result of some integer calculation produces a number that overflows 31 bits. That is, the class LargeInteger (or its proxy) should be available to the VM to instantiate the correct object. Additionally, large integers implement their arithmetic methods by calling primitives from external plugins (the large integers plugin).

Polymorphism. The introduction of large integers also introduces *polymorphism* between large and small integers. SmallInteger and LargePositiveInteger share a common superclass Integer. factorial is defined in the class Integer. Both subclasses define their own implementation of the arithmetic methods for addition and multiplication.

5.5 Experiment IV: Reflective invocations

The fourth experiment introduces reflective invocations. Figure 10 introduces the code we used for this experiment. The class User has fields name and age, and four methods. Two of these methods (age and name) are getters, the method hasWritePermissions is annotated as property (using a pragma in Pharo's terminology) [DMP16] and the method isMinor is a normal method. We also introduce the class PropertyExtractor with the responsibility of returning the name of those methods that are properties of an object *i.e.,* all getter methods, and all those methods annotated as property. The statement we introduced as the entry point for this experiment is "PropertyExtractor new extractPropertiesFrom: User new".

```
1  Object subclass: #User
2     instanceVariableNames: 'name age'.
3
4  User>>age (
5     ^ age )
6
7  User>>hasWritePermissions (
8     <property>
9     ^ true )
10
11 User>>name (
12    ^ name )
13
14 User>>isMinor (
15    ^ age < 18 )
16
17 PropertyExtractor>>extractPropertiesFrom: anObject (
18    ^ anObject class methods
19       select: [ :each | each isReturnField or: [ each pragmas anySatisfy: [ :pragma | pragma keyword =
         #property ] ] ]
20       thenCollect: [ :each | each selector ] )
```

Figure 10 – **Code of the reflective invocations experiment.** The PropertyExtractor class does the reflective invocations, User is the class we will be reflecting on.

This experiment evaluates how Tornado handles reflective invocations. The PropertyExtractor queries the methods from the User class, which are included as part of the tailored application (since they receive the messages isReturnField and pragmas). These reflective invocations include: (a) access to the class of an object, (b) access class methods and (c) query those methods to know if they correspond to the criteria of the PropertyExtractor.

5.6 Experiment V: Adding I/O

A fifth experiment introduces I/O to each of the previous experiments, adding a statement printing to the standard output the obtained results. Figure 11 shows the code from our entry point in the case of summing up two numbers. The entry points

for the other experiments have the same structure, differing only in the expression that is printed (the "1+2" expression in this case).

```
1 FileStream startUp: true.
2 FileStream stdout
3     nextPutAll: (1 + 2) asString;
4     crlf.
```

Figure 11 – **Entry point of the experiment that sums two numbers and prints the result in the standard output stream.**

Note also that we needed to include as part of the entry point the initialization of the class FileStream (FileStream startUp: true). This statement initializes the File library every time the program is started. Thus, this experiment evaluated the proper usage of I/O streams such as the standard output stream, and the ability of Tornado to handle *platform specific features*. Pharo is a platform independent language and thus some of its libraries (e.g., file management) have code specific to different platforms (*e.g.,* operating system, 32bits vs 64 bits). This experiment shows that Tornado prepares tailored versions of applications to run on a single operating system or platform.

5.7 Experiment VI: A Web Application

Our last experiment consists in tailoring a web application using the Seaside application framework [DLR07]. Seaside is a web application framework featuring continuations thanks to stack reification. We configured it with its default values, without making any customizations. The web application under tailoring has a single webpage that allows one to send requests to the web server to increase or decrease a counter. This experiment shows that Tornado works for applications requiring to launch and synchronize threads/processes. The Seaside application framework makes use of Pharo processes. One process listens to incoming connections and opens new processes to handle requests. Seaside uses semaphores to synchronize processes and wait for incoming data from sockets.

For this case, we set up two different experiments, with two different seeds. We first used the empty seed (*Seaside Web Application A*), as in the previous experiments, and then used a seed containing all Pharo base libraries (*Seaside Web Application B*). For reasons of space, the details of how the entry points are initialized for both seeds can be found in our technical report [PDBF11].

5.8 Results

We gathered our experimental results into Table 1. This table shows:

Experiment. The name of the experiment under evaluation, followed by our measurements.

Reference Application. The size of the *PharoKernel* application; and between parentheses the size of the official Pharo distribution prepared for production (cf. Section 5.1).

Seed. The size in KB of the chosen seed for the experiment.

Nurtured Application. The final size of the nurtured application extracted by Tornado.

Saved. The percentage of space saved using the smallest reference application size. We chose the smallest reference application to avoid biased results in our favor. We calculated this percentage using the following equation:

$$saved = 100 - \frac{100 * (nurtured - seed)}{reference - seed}$$

Note that we subtract the size of the seed from both the nurtured and reference application sizes, since the seed is shared between both. That way, we compare only those parts of the application that were subject to the RFG algorithm.

Experiment	**Reference App** ***Shrunk(Production)***	**Seed Size**	**Nurtured App**	**Installed Code**	**Saved(%)**
Sum Two Numbers (I)	3799 (12873)	10	11	1	**99.97%**
Factorial 10 (II)	3799 (12873)	10	15	5	**99.87%**
Factorial 100 (III)	3799 (12873)	10	18	8	**99.79%**
Reflective App (IV)	3799 (12873)	10	32	22	**99.42%**
(I) + I/O	3799 (12873)	10	81	71	**98.13%**
(II) + I/O	3799 (12873)	10	82	72	**98.10%**
(III) + I/O	3799 (12873)	10	89	79	**97.92%**
(IV) + I/O	3799 (12873)	10	95	85	**97.76%**
Seaside Web App A	20254 (17250)	10	573	563	**96.73%**
Seaside Web App B	20254 (17250)	12872	13090	218	**95.02%**

Table 1 – **Results of the tailored experiments.** Sizes are displayed in KB. The percentage of saved space does not take into account the seed, as it is not subject to Tornado and it is shared by both the reference and nurtured application.

Component	**Size (KB)**
Pharo Base Libraries	12872
Seaside Application Framework Libraries	4378
Seaside Web App	47
Reflective Invocations App	104

Table 2 – **Component sizes in our experiments. Size presented in KB.**

Table 2 shows the size in KB of the program units we used in our experiments. This table details the size of the Pharo base libraries, third-party libraries such as Seaside and our particular experiments, which aid in the understanding of the results. We obtained these sizes by measuring the size of the program units once loaded in memory.

Result Discussion. Our experiments show that Tornado aggressively reduces the size of program units required for an application. Our examples save from 95% to 99% of space, compared with their reference application (which contains all base libraries and third party libraries in case of Seaside).

Our first three experiments (the sum of two numbers, and the factorial of 10 and 100) show that Tornado succeeds to create minimal deployment versions of our applications, taking into account that our seed forces a minimal of 10KB in each of them. The reflective application is indeed also minimal, but bigger than the other three, as Tornado installs inside the nurtured application (a) all the code that is accessed by reflection and (b) code from the collections package to iterate the methods of a class.

We detect a notorious growth in size when adding I/O to our experiments, which varies from 63KB to 71KB extra. According to the list of installed program units, we identify a problem in the design of the Pharo I/O streams library: a set of character tables used for character encoding and conversion are initialized, even if not all of them are later on used by the application. This problem shows that this part of Pharo base libraries should be rethought to lazy initialize this data, or that we should improve Tornado's propagation rules with a more efficient mapping.

The web application (Seaside based) experiments show that Tornado can be used in a complex setting such as a web application that runs a web server, while still achieving good results. It is interesting to note, from the comparison of both experiments, that more of half of the size of the final nurtured application in *Seaside Web Application A* seems to be in the base libraries, as the amount of installed code is reduced when introducing the base libraries seed.

Comparison with a Dedicated Platform. To have a broader view of our results, we compare them to MicroSqueak [Mal]. MicroSqueak is a dedicated platform that runs on the Pharo platform *i.e.,* a specialized platform containing an alternative implementation of base libraries, as Java Micro Edition (J2ME) [Jav] is for Java. MicroSqueak was designed with the explicit goal to be the smallest practical Squeak kernel. It contains a total of 49 classes with a reduced set of methods. It offers a minimal core of the language, a basic collection library and basic file I/O support. MicroSqueak presents a minimal memory footprint of 80KB, when we build an application that performs no computation.

On one side, Tornado ensures smaller memory footprints when working on small applications. On the other side, MicroSqueak presents crucial differences with Pharo base libraries: it does not provide the same libraries (*e.g.,* it does not contain socket support) and it does not provide the same API of the libraries that it contains. Thus, applications such as the ones used in our Seaside experiment cannot run on top of MicroSqueak without a dedicated version of the Seaside framework.

6 Comparison of Tornado with Related Work

We start this section by evaluating Tornado according to the criteria we defined in Section 2.3, so we can discuss it and compare it to other existing approaches in the following sections.

6.1 Evaluation of Tornado

Table 3 shows an overview of existing families of solutions and Tornado, and how they map to the criteria defined in Section 2.3. In this section we focus on the evaluation of Tornado that is summarized in the latest column of the table.

Tornado's model and implementation show themselves as the most complete solution in the area of application tailoring. It tailors program units written by the application developer as well as those from the base language and third-party libraries. There is no special code for managing such cases since the infrastructure of Tornado allows the inspection of loaded classes, regardless of their origin. This approach, based on runtime execution, offers two main advantages: (a) it does not require modifications in the nurtured application code allowing its usage on legacy code and libraries in

a transparent way, and (b) it supports reflection naturally since the code exercised during tailoring is the same that will be executed once deployed.

Tornado requires a dedicated infrastructure only during tailoring: tools to monitor and manipulate the tailored application. However, once tailoring is finished, Tornado extracts and prepares the application to run in the deployment-ready unmodified infrastructure.

Finally, Tornado is a flexible solution in the sense that it allows one to configure the level of tailoring by means of a seed. The seed contains a preselection of program units available in the tailoring application before tailoring starts. In this way we can use the seed to specify whether, for example, the base or third-party libraries should be tailored or not.

	Dedicated platforms	**Static Analysis**	**Hybrid Analysis**	**Dynamic Analysis**	**RFG**
Reflection Awareness	+	-	-	+	+
Base Lib. Support	+	+	+	+	+
Third-Party Lib. Support	-	+	+	+	+
Legacy Code Support	-	+	+	+	+
Standard Deployment Infrastructure	-	+	-	-	+
Configurability	-	-	-	-	+
Applicability Without Type Annotations	-	-	-	+	+
Completeness	-	-	-	-	-

Reflection Support	**Base Library Support**	**Third-Party Library Support**	**Legacy Code Support**	**Dedicated Infrastructure for Deployment**	**Flexibility**	**Ensures Completeness**	**Applicability without Type Annotations**
(+) complete (−) partial	(+) supported (−) not supported	(+) supported (−) not supported	(+) supported (−) not supported	(+) not needed (−) needed	(+) configurable (−) fixed level	(+) yes (−) no	(+) supported (−) need complementary techniques

Table 3 – Evaluation criteria applied to tailoring techniques

The reduction of the deployment footprint of object-oriented applications has been a subject of interest both in industry and research since many years. We identified four different families of solutions for dead code elimination: dedicated platforms (cf. Section 6.2), static analyses (cf. Section 6.3), dynamic analyses (cf. Section 6.4) and hybrid analyses (cf. Section 6.5). Table 3 presents a comparison of these techniques, given the criteria defined in section 2.3.

6.2 Dedicated platforms

Dedicated platforms are platforms containing frameworks and/or libraries prepared to run under specific circumstances *e.g.,* Java Micro Edition (J2ME) [Jav] is the dedicated version of the Java platform, and Cocoa Touch is the one of Cocoa. These specialized platforms are reduced platforms to run applications inside mobile and constrained devices. These platforms provide a reduced and fixed set of base libraries defined a priori and in a non-customizable way. Applications have to be specially written for these platforms, and thus legacy code and third-party libraries are not compatible. Reflection is available since the statically tailored base libraries are built in a non-automated fashion. In other words, there is no tailoring process applied to the application code.

6.3 Static Analysis-Based Techniques

Static analysis approaches for dead code elimination make use of the static information of a program to select the minimal subset of used elements. Well-known techniques such as program slicing [Tip95] use static analyses to do dead code elimination with a statement granularity, transforming programs at a sub-method level. RFG differs from these solutions by using a more coarse granularity where the smallest elements we tailor are methods and single objects, followed by classes. The rest of this section discusses static analysis-based solutions with granularities similar to RFG's.

The literature describes four different algorithms to achieve application tailoring as described in this paper: unique name, class hierarchy analysis (CHA), rapid type analysis (RTA) and reachable members analysis (RMA) [BS96, Tit06]. These techniques share a common approach, selecting an entry point method of an application and following from it the execution flow using the available static information *i.e.,* type annotations, and class and method names, building a call-graph [GDDC97].

These techniques have been studied and applied in many environments and languages. Rayside et al. [RK02], Jax [TSL03] and the ExoVM System [Tit06] propose application extraction tools using these techniques for Java applications. Sallenave et al. [SD10] apply RTA to produce smaller .NET assemblies for embedded systems. Bournoutian et al. [BO14] use CHA to optimize on-device Objective-C applications. Ole Agesen [Age96] presents in his thesis a static technique applied to Self, a dynamically-typed language. Ole Agesen uses type inference to obtain type information and use it to select which objects to extract.

In summary, these approaches are based on the static types found either in the source code or bytecode, or recreated through type inference. They are not applicable *efficiently* in dynamic languages with no static type declarations. These solutions are valuable as they allow one to tailor base and third-party libraries, and legacy code. Their tailoring approach generates new deployment units that can run on the standard runtime infrastructure. The main drawback of this approach appears in the presence of reflection and configuration files, which will only work with a subset of reflective invocations through complementary analyses on the strings found in the source code [BSS+11]. Also, existing solutions in this family lack the flexibility to declare and identify levels of tailoring, making it an "all or nothing".

6.4 Dynamic Analysis-Based Techniques

Dynamic analysis techniques use exclusively runtime information (*i.e.,* execution flow, alive objects, execution statistics) to perform dead code elimination. Amongst these, we identify two different approaches: *load on demand* and *code collection.* Load on demand approaches detect during runtime whenever a class or method needs to be installed and request it to a server application. Code collection approaches deploy the full application and garbage collect unused code based on usage statistics. Related work in this family share a common characteristic: these techniques are used inside ubiquitous systems *i.e.,* systems meant to be always connected. Ubiquitous systems, as they are always connected, have a possibility to fallback and recover in the case of incompleteness. However, to focus here on the dead code elimination techniques, we will discuss the incompleteness recovery techniques in Section 7.

JUCE [PRT+04, TP01]. It is a platform for ubiquitous devices supporting code load on demand and code collection. Its approach for building up an application

is similar to Tornado. First, it initializes a minimal running application and code is loaded, with method granularity, from a server located in a different machine. Unused code is collected following usage statistics, and loaded back again on demand if needed.

OLIE [GNM+03]. It is an engine that intelligently partitions and offloads objects during runtime to minimize memory consumption. It is part of the adaptive infrastructure for distributed loading (AIDE). In OLIE, offloaded objects are indeed migrated to nearby remote devices. Migrated objects can be accessed later through proxies that perform remote invocations on them.

SlimVM [KWW+09, WGF11]. It is an ubiquitous system where all code resides on a remote server and is loaded only on demand on small devices. Some static analysis is performed only on the server to reduce the size of the transported code, by identifying most likely needed code and increasing the granularity of the transported code. SlimVM imposes a change of class format to deploy an application on it.

All solutions inside this category share one main property: they require to run the application inside a dedicated infrastructure to apply their techniques *e.g.,* dedicated VMs implementing remote lazy loading, code collection or new bytecode sets. The main challenge of these solutions resides in applying these techniques while minimizing their impact on performance during the runtime. Additionally, these solutions require their applications to run exclusively inside their infrastructure. Tornado works in the same way as these solutions: it uses a dedicated infrastructure to run the desired application and select the used elements. However, Tornado provides also the ability to extract this application and run in *offline* mode, using the non-modified infrastructure.

Regarding dynamic features such as reflection, these solutions are the ones that can, potentially, handle it in the best way since they have at runtime all the information needed to resolve it. JUCE and OLIE, as Tornado, handle naturally reflection as they do not change the runtime representation (an assumption made by programs using metaprogramming). SlimVM on the other side, had to change the reflection support because they changed the object and class representation on their VM.

Regarding its applicability, SlimVM needs to recompile the whole application into its own format, while OLIE and JUCE, as Tornado, can tailor base and third party libraries without any modifications on it. Thus, the latter two can be applied to legacy code also for free. None of these solutions offer the ability to select the level of tailoring which always work on the full application. In contrast, Tornado uses seeds to force a minimal subset of elements to be part of the application.

6.5 Hybrid Analysis-Based Techniques

Hybrid analysis techniques mix static and dynamic (*i.e.,* runtime) information to provide better results. The common approach of these is to start an application, such as Tornado does, and pause it after some minimal runtime information is available *i.e.,* call stacks are created, some classes are loaded and initialized, and some objects are instantiated. Then, it uses the built stack of alive objects to perform a static analysis, as described in Section 6.3, with concrete type information.

Java in The Small (JITS) [CGV10] uses a hybrid approach to select the used parts of a program, and then loads them inside a binary image. A dedicated VM loads the binary image at startup. The approach of JITS tailors base and third-party libraries

as well as application specific code. It does not require modifications on the existent application to tailor it, so a legacy application could theoretically be tailored with this approach. JITS does not offer the possibility to configure the tailoring level, since it was designed to be used only in embedded devices where no more than one application would be running. Regarding reflection, JITS presents the same drawbacks as the other static call graph analysis approaches since not all the runtime information about the reflective invocations can be deduced.

Partial Evaluation [JGS93] is a technique of program specialization by using abstract interpretation. Using partial evaluation, a program evaluated with respect to only a part of its expected input produces a residual program. This residual program is specialized with respect to the already received input and expects the rest of its input. There are two key differences between RFG/Tornado and partial evaluation. Conceptually speaking, the result of RFG is not a residual program that expects some residual input, it is a tailored application that can keep growing if we feed it with new inputs. Regarding the mechanics behind it, RFG requires the real execution of the program to capture the most dynamic properties of a program.

7 Discussions on the run-fail-grow approach

7.1 Ensuring Completeness

Dead code elimination techniques do never ensure code completeness by themselves. That is, they do not guarantee that the application does not contain extra program units, nor that it does not miss program units. Static approaches cannot efficiently predict the need of those elements used by reflection, or configured in external files/resources. Dynamic approaches depend on the code coverage of the application during runtime, *i.e.,* parts of the application that are not used will not be available afterwards. Hybrid approaches share both weaknesses. Orthogonal to the dead code elimination techniques, two complementary mechanisms are used by existing solutions to guarantee completeness and avoid runtime errors due to missing code.

Lazy Loading. JUCE [PRT+04, TP01] and SlimVM [KWW+09, WGF11] load missing code from remote servers on demand. Marea [PBD+13] is an implementation of an application-level virtual memory system with lazy loading of unloaded unused objects. These different solutions differ in their lazy loading approaches by the granularity they use. JUCE relies on method granularity to control memory consumption. SlimVM uses by default basic block granularity, but it can work at the class and method level too. Marea uses an object-cluster granularity. It loads object graphs containing not only classes but also individual objects, which were unloaded to reduce the applications memory footprint.

Remote Invocations. OLIE [GNM+03] uses remote invocations to invoke methods from those objects that where offloaded and migrated to other devices. This approach may introduce several latency problems due to network communications. OLIE tries to minimize it by offloading those elements that degrade less the performance of the system. For that, it records object and bandwidth usage statistics at runtime.

We could imagine implementing such strategies in Tornado by introducing two main ingredients:

1. Skip the application extraction step (Section 4.8). Keeping all traps alive keeps also Tornado's ability to detect and intercept the usage of missing program units.

2. Deploy and access the reference application remotely.

These two implementation variations would allow one to either keep Tornado's default lazy loading strategy or to implement a remote invocation strategy by changing the behavior of trap handlers. The trade-off of this solution is that it requires to deploy the application in a Tornado enabled environment and virtual machine, plus a remote reference application accessible trough the network.

7.2 Maximizing Run-Fail-Grow Effectiveness

Dynamic techniques, in particular Tornado, depend on the coverage of the application to ensure the code is loaded and available for execution. Application coverage must ensure that every program unit that is interesting to be deployed is covered, including special and boundary cases as well as the straightforward cases. We can enforce the coverage and installation of code with several techniques.

Manual Testing. Manual testing provides a simple but inefficient way to cover application code. Its main benefit is that program unit selection is based on user interactions. Its main drawback is the possibility of human omission during testing, which impacts directly the detection of used code.

Automated Testing. Automated testing counters the human omissions by adding repeatability in the generation of the deployment unit. Different levels of testing have different impacts on the coverage and will produce different results. For example, using unit testing to cover the application and library code may exercise more code than actually needed, since it usually tests smaller units and tends to cover the whole application/library. Acceptance tests may not exercise enough parts of the application. UI tests should be considered as part of the solution for maximizing coverage.

7.3 Application Design Supporting RFG

As shown in Section 5.2, the design of the tailored application directly impacts on the results obtained by Tornado. A series of issues appear regarding global state (*e.g.,* class variables and global variables).

A first issue is related to the initialization of such a global state [Ung95]. Since Tornado follows the application execution flow, eager initializations force Tornado to install objects and methods that may not be used later by the application. In contrast, lazy initializations will only be triggered on usage. Thus, better results could be obtained if a lazy initialization strategy were adopted for the global state.

A second issue appears with residual side-effects. Our tailoring technique builds the deployment application by running it. Thus, the executed global side-effects may reside in the tailored application. For example, a web application framework may hold a cache of HTTP sessions in a class variable. When the tailoring process finishes, the application will keep this cache if we do not handle the case. Solving this problem in Tornado may require either minimizing global state in an application, or either installing a new entry point to reinitialize such global state when the tailoring is finished *e.g.,* clean caches and session dependent state such as file and socket descriptors.

7.4 Modern Language Features

Tornado handles modern programming language features such as reflection, open classes and class extensions [BDW03] (*i.e.,* a package can add methods to classes from other packages) and traits [SDNB03], out of the box. Reflective invocations contain all the information they need to be tailored correctly as Tornado works during the runtime of the application. Tornado installs methods from other packages or behavior units such as traits seamlessly because during runtime it knows the exact concrete type of each object involved in the execution. Thus, no extra static or string analysis is needed. This is possible thanks to Ghost proxies [MPBD+11a], which can capture all message sends and specific method invocations.

7.5 Easily Managing Base libraries

Most applications do not use the whole base-library collection distributed along with a language. These libraries, representing big code code bases, are then potential candidates for removal. However, in most of the modern object-oriented languages, base language libraries are loaded and initialized by the language Virtual Machine (VM) as some times an order has to be ensured or those same program units are used internally by the VM. Thus, the application developer cannot easily manage and customize which of them she wants, since it often requires VM modifications.

Pharo provides the developer with access to the base libraries in the language. Thanks to this ability, Tornado can manage Pharo's base libraries as it manages application code. There is, however, an exception: the program units that belong to the interface between the language kernel (*i.e.,* the minimal language elements that should be available to run) and the VM must be installed and initialized in a particular order and be always present in the nurtured application. Because of this, we guarantee that the minimal seed, the *empty seed*, contains at least all these needed program units.

7.6 Alternatives for an RFG Implementation

Our RFG implementation, Tornado, is based on an architecture that allows complex manipulations on both the nurtured and the reference applications. These manipulations include introspective features, code installation, isolation, pointer swapping and so on. We can imagine alternative implementations of RFG that pose different requirements but not necessarily supporting the same features.

As a first variant, we could imagine traps being implemented through code-rewriting. We preferred a proxy-based implementation to these two because they fit more naturally in RFG's dynamic approach. Our choice also solves naturally the problem of managing identity. For example, let us consider two objects Alice and Bob that know a third object Charles. In Tornado, as long as Charles is not used, Alice and Bob will refer to a single proxy for Charles. When either Alice or Bob use Charles, Tornado installs a copy of the reference Charles object and replaces its proxy with it. Now both Alice and Bob refer to the same Charles object. Solving this problem with code rewriting would require to know all call sites where Charles could be used, and update them accordingly.

In a second variant used (and unused) objects can be tracked using a specialized interpreter, for example, in a modified virtual machine. Such an implementation requires to trace the entire program execution and mark used program units. Once

the execution is finished, we should recover all marked program units to generate the deployment artifact. This alternative solution does not require two isolated environments but simply re-executes or interprets the reference application. The trade-off of this solution is that the state of the nurtured application cannot be specialized for a special run, without altering the reference application. Indeed, the isolation mechanism in Tornado avoids interference between the reference and nurtured applications, and lets us guarantee the initial state of the nurtured application. For example, if we consider a reference application that has a filled global cache, we would like to ensure that cache to be empty in the nurtured application instead of transferring the cached values from the reference application.

8 Conclusion

In this paper we presented our run-fail-grow (RFG) approach for application tailoring. RFG tailors an application by starting it and initializing it with a seed that contains the minimal set of program units we want to be present in the tailored application. Then, we install and execute the application entry points. As the application executes, missing program units are found and installed on demand, ensuring that only the needed program units are introduced. By following the runtime execution, it supports dynamic features such as reflection and meta-programming.

We implemented RFG in a tool called Tornado, which succeeds to produce applications with minimal footprint for deployment. Our results show that we manage different extreme and challenging cases with flexibility.

We see three different evolution paths of this work: we would like to first, study the mechanisms that could be used to ensure completeness; second, study the usage of this approach in the context of dynamic adaptation and update of applications; and third, extend this work to tailor virtual machines and system libraries.

References

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. Ph.D. thesis, Stanford University, December 1996.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. URL: `http://pharobyexample.org/`,`http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf`.

[BDW03] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003. Best Paper Award. `doi:10.1007/b12023`.

[BNGG11] Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, and Manish Gupta. *Software*, pages 408–432. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. `doi:10.1007/978-3-642-22655-7_19`.

[BO14] Garo Bournoutian and Alex Orailoglu. On-device objective-c application optimization framework for high-performance mobile pro-

cessors. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 85:1–85:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association. URL: http://dl.acm.org/citation.cfm?id=2616606.2616711.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM. doi:10.1145/236337.236371.

[BSS+11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM. doi:10.1145/1985793.1985827.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press. URL: http://bracha.org/mirrors.pdf.

[CGV10] Alexandre Courbot, Gilles Grimaud, and Jean-Jacques Vandewalle. Efficient off-board deployment and customization of virtual machine-based embedded systems. *ACM Transaction on Embedded Computer Systems*, 9:21:1–21:53, mar 2010. doi:10.1145/1698772.1698779.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 83–100, New York, NY, USA, 1996. ACM. doi:10.1145/236337.236344.

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007. URL: http://rmod.inria.fr/archives/papers/Duca07a-IEEESoftware-Seaside.pdf, doi:10.1109/MS.2007.144.

[DMP16] Stéphane Ducasse, Eliot Miranda, and Alain Plantec. Pragmas: Literal Messages as Powerful Method Annotations. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, August 2016. URL: http://rmod.inria.fr/archives/papers/Duca16a-Pragmas-IWST.pdf, doi:10.1145/2991041.2991050.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM. doi:10.1145/263698.264352.

[GHVJ93] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag. URL: `ftp://st.cs.uiuc.edu/pub/papers/patterns/ecoop93-patterns.ps`.

[GNM+03] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, PERCOM '03, pages 107–, Washington, DC, USA, 2003. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=826025.826367`.

[Jav] Java micro edition. `http://java.sun.com/javame/index.jsp`.

[JGS93] Neil J. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[KWW+09] Christoph Kerschbaumer, Gregor Wagner, Christian Wimmer, Andreas Gal, Christian Steger, and Michael Franz. SlimVM: A Small Footprint Java Virtual Machine for Connected Embedded Systems. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 133–142, New York, NY, USA, 2009. ACM. `doi:10.1145/1596655.1596678`.

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *Proceedings of Asian Symposium on Programming Languages and Systems*, 2005.

[Mal] John Maloney. Microsqueak. `http://web.media.mit.edu/~jmaloney/microsqueak/`.

[MP12] Mariano Martinez Peck. *Application-Level Virtual Memory for Object-Oriented Systems*. PhD thesis, Ecole des Mines de Douai - France & Université Lille 1 - France, October 2012. URL: `http://tel.archives-ouvertes.fr/docs/00/76/49/91/PDF/PhD_-_Mariano_Martinez_Peck.pdf`.

[MPBD+10] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Visualizing objects and memory usage. In *Smalltalks 2010*, Concepción del Uruguay, Argentina, 2010. URL: `http://rmod.inria.fr/archives/workshops/Mart10a-Smalltalks2010-VisualizingUnusedObjects.pdf`.

[MPBD+11a] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Efficient proxies in Smalltalk. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'11)*, Edinburgh, Scotland, 2011. `doi:10.1145/2166929.2166937`.

[MPBD+11b] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Problems and challenges when building a manager for unused objects. In *Proceedings of Smalltalks 2011 International Workshop*, Bernal, Buenos Aires, Argentina,

2011. URL: http://rmod.inria.fr/archives/workshops/Mart11b-Smalltalks2011-UOM.pdf.

[PBD+13] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Marea: An efficient application-level object graph swapper. *Journal of Object Technology*, 12(1):2:1–30, jan 2013. doi:10.5381/jot.2013.12.1.a2.

[PBF+15] Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Marcus Denker, and Camille Teruel. Ghost: A uniform and general-purpose proxy implementation. *Journal of Object Technology*, 98:339–359, 2015. doi:10.1016/j.scico.2014.05.015.

[PDBF11] Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, and Luc Fabresse. Extended results of Tornado: A Run-Fail-Grow approach for Dynamic Application Tayloring. Technical report, RMod – INRIA Lille-Nord Europe, 2011. URL: http://rmod.inria.fr/archives/reports/Poli14-TechReport-Tornado-INRIA.pdf.

[PDF+14] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin van Ryseghem. Bootstrapping reflective systems: The case of pharo. *Science of Computer Programming*, 2014. URL: http://rmod.inria.fr/archives/papers/Poli14c-BootstrappingASmalltalk-ScienceOfComputerProgramming.pdf.

[Pol15] Guillermo Polito. *Virtualization Support for Application Runtime Specialization and Extension.* PhD thesis, University Lille 1 - Sciences et Technologies - France, April 2015. URL: http://rmod.inria.fr/archives/phd/PhD-2015-Polito.pdf.

[PRT+04] Lucian Popa, Costin Raiciu, Radu Teodorescu, Irina Athanasiu, and Raju Pandey. Using code collection to support large applications on mobile devices. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, MobiCom '04, pages 16–29, New York, NY, USA, 2004. ACM. doi:10.1145/1023720.1023723.

[RK02] Derek Rayside and Kostas Kontogiannis. Extracting Java Library Subsets for Deployment on Embedded Systems. *Sci. Comput. Program.*, 45(2-3):245–270, November 2002. doi:10.1016/S0167-6423(02)00059-X.

[SD10] Olivier Sallenave and Roland Ducournau. Efficient compilation of .net programs for embedded systems. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM. doi:10.1145/1925801.1925804.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003. doi:10.1007/b11832.

[TGP89] David Taenzer, Murthy Ganti, and Sunil Podar. Problems in object-oriented software reuse. In S. Cook, editor, *Proceedings ECOOP '89*, pages 25–38, Nottingham, July 1989. Cambridge University Press.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[Tit06] Ben L. Titzer. Virgil: objects on the head of a pin. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 191–208, New York, NY, USA, 2006. ACM. doi:10.1145/1167473.1167489.

[TP01] Radu Teodorescu and Raju Pandey. Using JIT Compilation and Configurable Runtime Systems for Efficient Deployment of Java Programs on Ubiquitous Devices. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, UbiComp '01, pages 76–95, London, UK, UK, 2001. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=647987.741339.

[TSL03] Frank Tip, Peter F. Sweeney, and Chris Laffra. Extracting Library-based Java Applications. *Commun. ACM*, 46(8):35–40, August 2003. doi:10.1145/859670.859695.

[Ung95] David Ungar. Annotating objects for transport to other worlds. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 73–87, New York, NY, USA, 1995. ACM. doi:10.1145/217838.217845.

[VBLN11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, pages 959–972, New York, NY, USA, 2011. ACM. doi:10.1145/2048066.2048138.

[VCM10] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Dynamic Language Symposium*, volume 45, pages 59–72. ACM, oct 2010. doi:10.1145/1899661.1869638.

[WGF11] Gregor Wagner, Andreas Gal, and Michael Franz. "Slimming" a Java Virtual Machine by Way of Cold Code Removal and Optimistic Partial Program Loading. *Sci. Comput. Program.*, 76(11):1037–1053, November 2011. doi:10.1016/j.scico.2010.04.008.

[XMA+10] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-scale Object-oriented Applications. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 421–426, New York, NY, USA, 2010. ACM. doi:10.1145/1882362.1882448.

About the authors

G. Polito is research engineer at CNRS working currently in the RMoD (http://rmod.lille.inria.fr) and Emeraude (http://www.cristal.univ-lille.fr/emeraude/) teams. His research targets programming language abstractions and tool support for modular long-lived systems. For this, he studies how reflective systems can evolve while maintaining these properties. He is interested in how these concepts combine with distribution and concurrency.

L. Fabresse is associate professor in the CAR research theme (http://car.mines-douai.fr) at the Mines-Telecom Institute, Mines Douai, France. His researches aims at easing the development of mobile and constrained software using dynamic and reflective languages such as Pharo. One of his goal is to support live programming of mobile and autonomous robots in an efficient way. He is the co-author of multiple research papers (http://car.mines-douai.fr/luc) and he concretizes all these ideas (models and tools) in the PhaROS plateform (a Pharo client for the Robotics Operating System) to develop, debug, test, deploy, execute and benchmark robotics applications. Each year, Luc also gives computer science lectures, co-organizes events (technical days, conferences, ...) and promotes Smalltalk as an ESUG (European Smalltlak User Group) board member.

N. Bouraqadi is a full professor at the Mines-Telecom Institute, Mines Douai, France, where he leads the CAR team (http://car.mines-douai.fr). His research targets mobile and autonomous robots from two complementary perspectives: Software Engineering (SE) and (AI). From the SE perspective, he studies software architectures, languages and tools for controlling individual robots. He advocates reflective and dynamic languages for modular and agile development of robotic software architectures. From the AI perspective, he studies coordination and cooperation in robotic fleets. He is interested in communication models as well as organizations for multi-agent robotic systems.

S. Ducasse is directeur de recherche at Inria. He leads the RMoD (http://rmod.lille.inria.fr) team. He is expert in two domains: object-oriented language design and reengineering. He worked on traits, composable groups of methods. Traits have been introduced in Pharo, Perl, PHP and under a variant into Scala, Fortress of SUN Microsystems. He is also expert on software quality, program understanding, program visualisations, reengineering and meta-modeling. He is one of the developer of Moose, an open-source software analysis platform http://www.moosetechnology.org/. He created http://www.synectique.eu/ a company building dedicated tools for advanced software analysis. He is one of the leader of Pharo (http://www.pharo.project.org/) a dynamic reflective object-oriented language supporting live programming. The objective of Pharo is to create an ecosystem where innovation and business bloom. He wrote several books such as Functional Programming in Scheme, Pharo by Example, Deep into Pharo, Object-oriented Reengineering Patterns, Dynamic web development with Seaside.